DTIC FILE COPY

(4)

AD-A203 087

THE VIRTUAL TIME MACHINE

Richard M. Fujimoto[1]
Computer Science Department
University of Utah
Salt Lake City, UT 84112
Technical Report Number UUCS-88-019

January 10, 1989

DTIC
ELECTE
JAN 2 6 1989
S D
D

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

[1]This work was supported by ONR Contract Number 00014-87-K-0184 and NSF grant DCR-8501826.

89 1 25 058

Abstract

Existing multiprocessors and multicomputers require the programmer or compiler to perform data dependence analysis at *compile time*. We propose a parallel computer that performs this task at *runtime*. In particular, the *Virtual Time Machine* (VTM) detects violations of data dependence constraints as they occur, and automatically recovers from them. A sophisticated memory system that is addressed using both a spatial and a *temporal* coordinate is used to efficiently implement this mechanism. Initially targeted for discrete event simulation applications, many of the ideas used in the machine architecture have direct application in the more general realm of parallel computation. The long term goal of this work is to develop a general purpose parallel computer that will support a wide range of parallel programming paradigms.

This paper outlines the motivations behind the VTM architecture, the underlying computation model, a proposed implementation, and initial performance results. A recurring theme that pervades the entire paper is our contention that existing shared memory and message-base machines do not pay adequate attention to the dimension of *time*. We argue that this architectural deficiency is the underlying reason behind many difficult problems in parallel computation today.

Accesion For
NTIS CRA&I
DTIC TAB
Unannounced
Justification
By per NP
Distribution /
Availability Codes
Dist | Avail and / or Special
A-1

1 Introduction

The difficulty of effectively utilizing existing parallel computers is well known. Although many successes have been achieved, the development of software for parallel machines that achieves performance significantly greater than a single processor implementation is far from routine, and requires considerably more effort than development of sequential code. We argue that many of the essential problems in parallel computation are exposed by the following observations:

Observation 1 *Parallel computation is essentially a sequencing problem.*

At the heart of the parallel computation problem is deciding what pieces of the program should be executed when. Specifically, one must adhere to certain data dependence constraints of the desired computation — if computation A produces a result that is required by computation B, then a *sequencing constraint* exists that A must be performed before B. The trick is to sequence the program so that it not only avoids violating these sequencing constraints, but it also exploits as much *concurrency* as the program and machine resources permit. We do not claim that sequencing is the *only* problem in parallel computation, but certainly it is important, and for many applications (discrete event simulation, for instance), very difficult.

Observation 2 *Current practice is to leave the sequencing problem to the compiler, or worse, the programmer to solve.*

In particular, the hardware provides little assistance. Typically, only a few synchronization primitives are provided (test-and-set, message passing, etc.), and the programmer/compiler is left to its own devices to figure out how to use them. We argue that solving the sequencing problem at compile time (or earlier) is fundamentally the wrong approach for many applications because:

Observation 3 *In general, sequencing constraints are not known until runtime.*

The data dependence relationships of the desired computation depend on the outcome of conditional branches, array index calculations, and dereferenced pointers, none of which are known at compile time. Also, interrupts and time slicing can greatly affect the temporal characteristics of the program in unpredictable ways.

These observations lead us to the conclusion that *parallel computation is hard because one must use static information to solve dynamic problems.* Programmers and compilers are ill equipped

for attacking the sequencing problem because they may only use information that is available at compile time, but sequencing constraints are fundamentally dynamic in nature.

We argue that it should be the *machine's* responsibility to guarantee proper sequencing. Only the machine can have ready access to the necessary information. Further, only the machine can provide the necessary muscle (hardware) to see to it that the sequencing problem is solved in an efficient way.

The deficiencies of existing machines become painfully clear when one attempts to parallelize discrete event simulation (DES) programs. There, data dependence relationships depend entirely on event timestamps that are computed throughout the course of the computation. Compile-time solutions are all but useless under these circumstances, and development of effective runtime strategies has also proven to be extremely difficult. All existing approaches using conventional multiprocessors have major liabilities and limitations, even for problems with substantial amounts of parallelism.[1] Simple operations, such as examining the value of a remote state variable, become difficult and time consuming, even on machines supporting shared memory. We attribute this dismal state of affairs to the fact that conventional machines provide no help in solving the sequencing problem.

Returning to machine architecture, it is reasonable to ask: why can't existing machines, or some minor variation thereof, ensure proper sequencing at runtime? We believe the fundamental problem is rooted in our final observation:

Observation 4 *The state of existing shared-memory and message-based machines contains virtually no* temporal *information.*[2]

Guaranteeing proper sequencing amounts to ensuring that the real-time sequence in which computations are performed is consistent with the data dependence relationships of the desired computation. This is difficult to achieve in existing machines because neither the constraints nor the temporal relationships among different portions of the computation (e.g., accesses to shared variables) are represented explicitly by the machine architecture in a usable form. As a consequence.

[1] Specifically, existing conservative mechanisms perform poorly if the application contains poor lookahead properties or if the average number of unprocessed events is small relative to the connectivity of the network topology [Fuj87], and optimistic methods suffer from extensive state saving overheads for applications with large amounts of system state [Fuj89].

[2] The little temporal information that is present is not available in a form that can be readily used for the problem at hand. For example, the real time clock is treated as an I/O device rather than a fundamental component of the machine architecture.

it is virtually impossible for the hardware to detect violation of data dependence constraints, let alone recover from them. The lack of ready access to this essential information prevents conventional machines from seriously addressing the sequencing problem.

We contend that parallel machines that explicitly represent *time* as a fundamental aspect of the machine architecture will provide significant advantage over those that do not. In particular, the *Virtual Time Machine* uses a space-time memory system that is addressed using both a spatial and a temporal component. As we shall soon see, the temporal coordinate is used to specify the precedence relationships between distinct units of the computation (tasks). Because temporal aspects of the computation are explicitly represented in the machine architecture, the hardware can detect violations of data dependence constraints as they occur, and automatically recover from them using a *rollback* mechanism. We believe that these aspects of the machine architecture will enable solution of difficult problems in parallel computation that cannot be solved using existing parallel computers.

In the near term, we expect that the Virtual Time Machine will enable effective parallelization of discrete event simulation problems that today cannot be solved using conventional machines. In the long term, we plan to use these ideas in a general purpose parallel computer that can effectively exploit parallelism in a wide range of programming paradigms, ranging from automatic parallelization of "dusty deck" FORTRAN code to modern object-oriented programs. More will be said about this later.

A key factor that distinguishes the VTM architecture from other approaches (e.g., data flow) is the emphasis on *optimistic* concurrency control. Empirical data suggest that optimistic algorithms provide significant advantage over conservative methods in many discrete event simulation problems [Fuj89]. We suspect that this will also be the case in many other problem domains.

Throughout this paper, we assume that a shared memory organization is used. This could, in principle, be implemented on top of a message-based machine architecture, as discussed in [Li88]. However, the underlying communication mechanism must be relatively efficient for the proposed approach to succeed.

The remainder of this paper is organized as follows: After we review related work, the computation model implemented by the Virtual Time Machine is described. We then outline a proposed implementation; in particular, we focus attention on the space-time memory system which is at the heart of the machine design. Next, we discuss initial performance results to support our claim that

this approach shows good promise for attacking difficult problems in parallel discrete event simulation. Finally, we briefly discuss the application of this approach to the automatic parallelization of sequential programs, and outline some important open questions that must still be addressed.

2 Related Work

The seminal work and underlying theory for the proposed architecture lies in Jefferson's theory of Virtual Time [Jef85]. The optimistic nature of the Time Warp synchronization mechanism (one implementation of Virtual Time) is a critical attribute of the VTM architecture that allows it to exploit parallelism in places where this might not otherwise be possible. This is similar to optimistic concurrency control in data base systems [KR81], except here, rollback is required to recover from synchronization faults.

ParaTran is a system that was independently developed by Tinker and Katz for automatically parallelizing sequential LISP programs [TK88,Tin88]. Because ParaTran also finds its origins in Time Warp, its computation model has many similarities to the one that is proposed here. However, ParaTran uses a software-based implementation (which is reported to be rather slow), and thus far, has only discussed hardware implementation at an abstract level. It remains to be seen if the ParaTran computation model can be efficiently implemented in hardware. We feel that hardware support is essential to achieve an efficient implementation of these mechanisms, and must be considered from the outset.

Knight does propose using special purpose hardware to aid the parallelization of sequential LISP code, but does not use Virtual Time to aggressively exploit parallelism [Kni86]. Cleary proposes using Virtual Time to parallelize PROLOG programs [CUL88] but no hardware support has yet been proposed.

The central distinction between the proposed architecture and that of existing special purpose simulation engines is that VTM is based on an *event oriented* simulation paradigm. Machines such as the Yorktown Simulation Engine [Pfi82] and commercially available logic simulation hardware are based on time-stepped methods. The inherent limitations of time-stepped methods in many problem domains is well known.

Finally, much of the VTM architecture is an extension of our earlier work on the rollback chip, which provides special purpose hardware to support Time Warp [FTG88a] (a slightly older version of the rollback chip is described in [FTG88b]). To our knowledge, our work on the rollback chip and

here on the Virtual Time Machine represent the first attempts to utilize special purpose hardware to support parallel discrete event simulation in a general way.

3 The Computation Model

The principal components of the computation model are *tasks* $C_0, C_1, ...$[3] and global[4] *data objects* (or simply objects) $O_0, O_1, ...$ The task is the indivisible unit of computation in the sense that one cannot roll back to the middle of a task. Similarly, the object is the atomic unit of state that can be restored via a rollback operation; data dependence analysis is only performed on an object as a whole (rather than individual words within the object). In principle, tasks and objects could be arbitrarily fine grained, but for the proposed implementation, medium grained tasks (e.g. a procedure or an iteration of a for-loop) and medium to large grained objects (e.g., a few hundred bytes to megabytes) are envisioned.

Tasks may also have local variables that are created when the task is instantiated (or is restarted after a rollback), and destroyed when execution is completed. Local state is not directly accessible to other tasks.

Each task is assigned a *unique* virtual time. This assignment specifies the precedence relationships among tasks. If task C_i generates a result (by storing it in some object) that is used by C_j, then C_i must be assigned a timestamp that is less than C_j's. If C_i and C_j do not directly or indirectly affect each other, no constraint exists concerning the relationship of their respective timestamps (except they should not be identical).

Tasks may (1) read and write local variables, (2) read and write global data objects, and (3) create new tasks. Here, the latter two operations are of greatest interest because accesses to local variables can be implemented using conventional techniques.

Each task must be executed in such a way that no data dependence violations occur *within* that task; the space-time memory described later only deals with data dependence violations *between* tasks. In VTM, tasks execute sequentially on a conventional processor. Well known techniques, such as instruction level parallelism and pipelining, may be used to exploit parallelism within each task.

The computation model contains two distinct data structures: the *state history* and the *execu-*

[3] C denotes "computation"; we reserve the T mnemonic for virtual time.

[4] Of course, one may use data abstraction techniques to restrict access to objects, but the machine makes no such a priori assumptions.

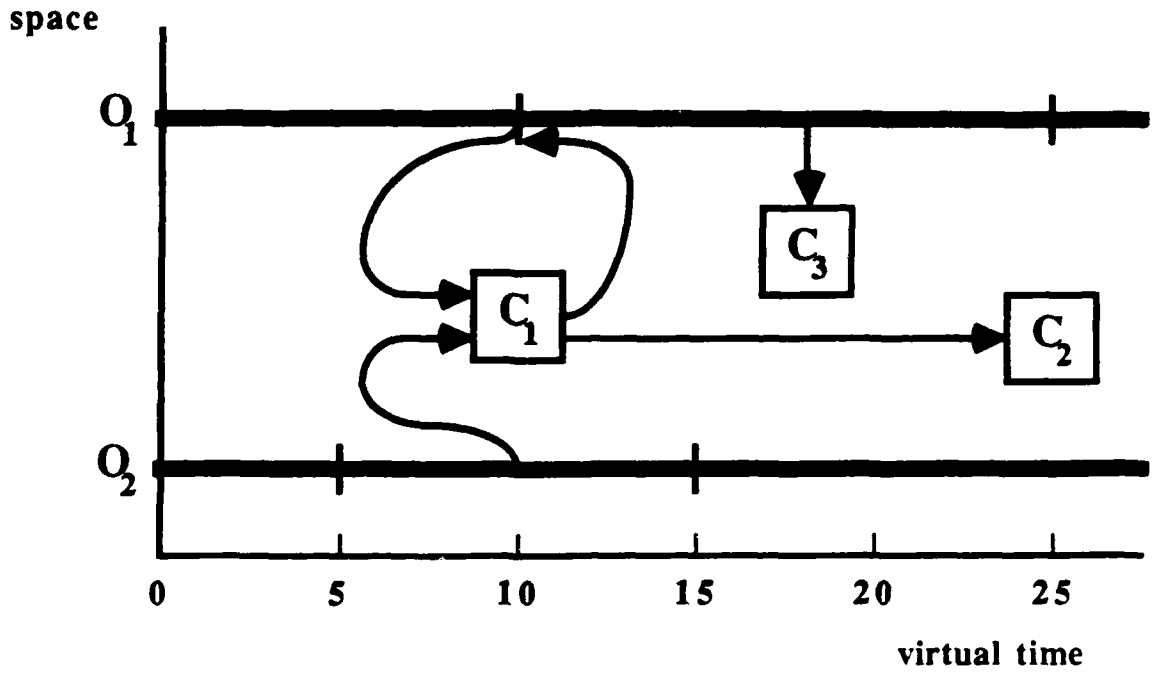

Figure 1: A space-time diagram is used to represent the computation model implemented by the Virtual Time Machine.

tion history. In addition, a *control mechanism* defines how these data structures are created and manipulated. We describe each of these next.

3.1 The State History

The state history represents the evolving state of data objects as the computation unfolds. It is convenient to represent the state history graphically by a *space-time diagram* such as that shown in figure 1. A point (t, O_i) in this diagram indicates the state of object O_i at virtual time t.[5] The evolving state of an object is represented by a horizontal *object history* line. The short vertical bars intersecting the object history denote *state changes*. For example, in figure 1, object O_1 changes state at virtual times 0 (where it was initialized), 10 and 25.

A segment of the object history line that is terminated at each end by state change bars, and is not broken by any other change bars, is called a *version* of the object; it represents the object's value over some virtual time interval. The temporal coordinate of the left state change bar indicates the time at which the version was created. In figure 1, there are three versions of object O_1 that cover the time intervals $[0, 10)$, $[10, 25)$, and $[25, \infty)$.[6]

[5] The horizontal axis is continuous and totally ordered, and the vertical axis is discrete and unordered.

[6] $[a, b)$ denotes the interval including a, but excluding b.

In the Virtual Time Machine, the state history is implemented by the *space-time* memory system. The history for a single object is implemented as a *stack* of that object's versions. Conceptually, creation of a new version can be viewed as copying the version currently on top of the stack into storage for a new version, and pushing this new version onto the stack. We will later describe an efficient implementation of this mechanism that avoids excessive copying.

3.2 The Execution History

The *execution history* is also represented on the space-time diagram. It consists of (1) task nodes, (2) data dependence arcs, and (3) causality arcs. Each *task node* represents the execution of a single task, and is placed on the graph at the virtual time coordinate assigned to that task; the spatial coordinate at which the task node is depicted is not significant. A *data dependency arc* extending from space-time coordinate (t, O_i) to task node C_k indicates that C_k examines some portion of O_i at virtual time t using one or more read operations. Similarly, a data dependency arc from C_k to (t, O_i) indicates that C_k modifies the state of O_i one or more times at virtual time t. This implies that a new version of O_i was created at time t, so a state change bar exists at point (t, O_i). For example, in figure 1, task C_1 (at time 10) examines objects O_1 and O_2 and modifies object O_1. The fact that the read dependence arc for O_1 is to the left of the state change bar is significant; it signifies the fact that the task examined the version of O_1 *before* modifying it. Had the task first modified the object, and *then* read it. the dependency arc would have been to the right of the state change bar.

Here, we assume that a task that is assigned virtual time T can only examine and modify objects at virtual time T.[7] A task may perform an arbitrary number of read and write operations on any object(s). and in any sequence (subject to adherence to the data dependence constraints within the task, as was discussed earlier). However, if tasks exhibit poor spatial locality (e.g., if each task modifies every object defined by the program) performance will be poor.

A *causality arc* extending from one task to another (e.g., C_1 to C_2) indicates that C_1 created task C_2. We assume causality arcs may only extend to the right in the space time diagram, i.e., if task C_i creates another task C_j, then C_j's virtual time must be strictly greater than C_i's. This eliminates the possibility of unending "rollback loops" or the "domino effect."

[7] The utility and consequences of relaxing this constraint is currently an open question.

3.3 The Control Mechanism

The control mechanism is responsible for constructing the state and execution histories. To simplify the discussion that follows, we will assume the existence of a global scheduling queue. Whenever a processor becomes idle, it removes the next task from this queue and executes it. In practice, this could be implemented as several independent queues to avoid a potential bottleneck.

The control mechanism was defined hand-in-hand with a proposed hardware realization in order to ensure the existence of an efficient implementation. This resulted in some compromises in the control mechanism, as will be discussed later. It is instructive to compare the model proposed here with that of ParaTran, where thus far, hardware realization has only been considered abstractly [TK88].

Tasks may invoke the following operations:

NEWTASK(T,C) creates the task C at virtual time T. C contains (at least) a pointer to the code for the task.

READ(T,S): D reads the version of data D at spatial address S and temporal address T from the space-time memory. S indicates both a data object and a word (or longword, byte, etc.) within that object.

WRITE(T,S,D) writes the data D at spatial address S and temporal address T. If this is the first time the task is writing into that object, a new version is created; otherwise, the value that was previously stored at that space-time address is overwritten.

The NEWTASK operation creates a record describing the task, and places it into the global scheduling queue. It is equivalent to scheduling an event in a discrete event simulation.

The READ operation performs a memory read of the space time memory. A data dependence arc must be created from the version being read to the task performing the read, if one does not already exist. This could be implemented by defining a record for each version listing all of the tasks that have accessed that version. Write operations are similarly logged. Exactly one write dependence arc exists for each version; this arc is a pointer to the task that created that version.

In the Virtual Time Machine, the logging of data dependence arcs is performed by hardware in each space-time memory module that watches references to that memory module. The processor need not wait for the logging operation to be complete, so we do not expect logging will degrade

performance. Alternatively, the logging could be performed when the task begins execution if the compiler and runtime system can determine in advance which objects will be referenced by the task.

The WRITE operation may trigger the violation of one or more data dependence constraints, and initiate rollback operations. For example, in figure 1, if some new task C_4 now writes to object O_2 at time 8, then C_1 must be rolled back because it previously read an erroneous value.

Suppose a task performs a write to object O_i at virtual time T. This will create a new version of the object if this is the first time the task is writing into the object;[8] otherwise, the write simply stores new data into the appropriate version. Any task that read O_i at a virtual time that is strictly greater than T is in danger of having read an erroneous value, so these tasks are rolled back.

In order to obtain an efficient implementation, we make the restriction that a write to object O_i may only access the version on top of O_i's object history stack (possibly after creating a new version and pushing it on top of the stack). We will explain the reason for this restriction later.

Whenever a task at virtual time T writes into an object, all versions of that object (if any) that were created at a time greater than T are first popped from the stack. A new version is created with timestamp T if one doesn't already exist, and the data are written into this version. Finally, all tasks that accessed the object at a time greater than T must be rolled back, as described below. In particular, tasks referencing versions popped from the stack must always be rolled back. In addition, if the current write operation created a new version, tasks that read the version that is immediately below the newly created one must be rolled back if their timestamp is strictly greater than T.

Rolling back a task essentially amounts to traversing a directed graph to roll back the tasks that were either directly or indirectly affected by the original, rolled back task. The causality and data dependence arcs form the links of this graph. Versions must also be popped from state history stacks to undo modifications of objects. More precisely, rolling back task C_i requires one to:

1. *retract* the writes performed by C_i,
2. *cancel* the tasks that were scheduled by C_i, and
3. return C_i to the scheduling queue.

[8]Recall that the time coordinate of write operations is assumed to be the virtual time assigned to the task, and these virtual times are assumed to be unique.

A write operation is *retracted* by popping the version created by the write, and newer ones above it, from the stack of the referenced object. Tasks that accessed these popped versions must also be rolled back.

Canceling a task is implemented by performing only steps (1) and (2) above, if the task has already been executed, or by simply deleting it from the scheduling queue if it has not. If the task is currently being executed, it must be aborted, e.g., via an interrupt mechanism, and then cancelled in the same way as tasks that executed to completion. The above mechanism is similar to *direct cancellation*, which is described elsewhere [Fuj89].

For example, in figure 1, a write to O_2 at time 8 will roll back C_1. This will cause the cancellation of C_2, and retraction of C_1's write to O_1. The latter will, in turn, cause the roll back of C_3.

The *lazy cancellation* policy proposed for Time Warp [Gaf88] can also be used. Lazy cancellation defers the cancellation of a task until it has been determined that the re-execution of the rolled back task does *not* recreate a task that is identical to the cancelled one.

4 More About the Computation Model

As mentioned earlier, the computation model described above represents a compromise between avoidance of unnecessary rollbacks and efficiency of implementation. In particular, the computation model will roll back tasks in certain situations where rollback is not strictly necessary. For example, because data dependence analysis is performed on a *per object* basis, a rollback could occur when out-of-order references are made to *different* variables within the same object.

The decision to implement data dependence analysis at an object-level granularity was made, in part, to keep the amount of associated overhead at a manageable level. Implementation of data dependence analysis *within* an object requires one to maintain for each task a record of all of the variables accessed by that task. The additional memory that is required to implement such a mechanism appears to be excessive.

Rather than trying to avoid unnecessary rollbacks in the hardware, we attack this problem at a higher level. Data dependence analysis using object-level granularity works about as well as analysis at a variable-level granularity if one can encapsulate the variables modified by a task (and no others) into a single object. Therefore, we rely on the compiler to partition the state space of the program into objects that maximize spatial locality, i.e., the compiler should cluster variables that are accessed by a task (and few others) into a single object. The effectiveness of this approach

has yet to be evaluated. We note, however, that the partitioning problem is simplified in certain programming paradigms, e.g., object-oriented languages.

A second, somewhat related, type of unnecessary rollback may also result. Suppose a variable A is written at virtual time 5, and read at virtual time 10. It is clear that any write to A at a virtual time preceding 5 will not invalidate the read at time 10. However, the model described above will roll back this computation.

In order to avoid this second type of unnecessary rollback, one would again have to maintain log information for individual words within the object, rather than the object as a whole. To see this, consider a second scenario that could arise: another variable B in the same object as A was written at virtual time 5 (rather than A in the previous scenario), A is then read at time 10, and finally, A was written at a time earlier than 5. This scenario *does* require a rollback. However, the hardware cannot distinguish this situation from the previous one unless it has log information distinguishing accesses to A from accesses to B. In addition, avoidance of this type of rollback would also require the hardware to be able to insert new versions into the middle of the stack. Although this can be done, it does reduce the efficiency of the implementation, and adds a nontrivial amount of complexity to the hardware.

Using Kuck's terminology [Kuc78], violation of data output-dependent constraints (WRITE followed by WRITE), as well as *data dependent* constraints (WRITE followed by READ), results in rollback in the Virtual Time Machine. However, violation of *anti-dependent* (READ followed by WRITE) constraints does *not* initiate rollback; one simply reads the old version. In contrast, no violation of any of these constraints is permitted in conventional machines.

If it is the case that tasks usually read data from an object before modifying it, then few unnecessary rollbacks due to data output-dependence constraint violations will occur. This is because a string of data dependence relationships will extend up the version stack, so a violation in the middle necessitates rolling back computations associated with versions higher in the stack. This read-modify-write scenario is typical in discrete event simulations constructed as a collection of concurrent processes.

Finally, we note that discrete event simulation problems map very naturally onto the computation model described above. In particular, the model is essentially a parallelized version of the familiar event oriented simulation mechanism that is at the heart of virtually all sequential discrete event simulation packages. Each task corresponds to a simulator event, data objects to state vari-

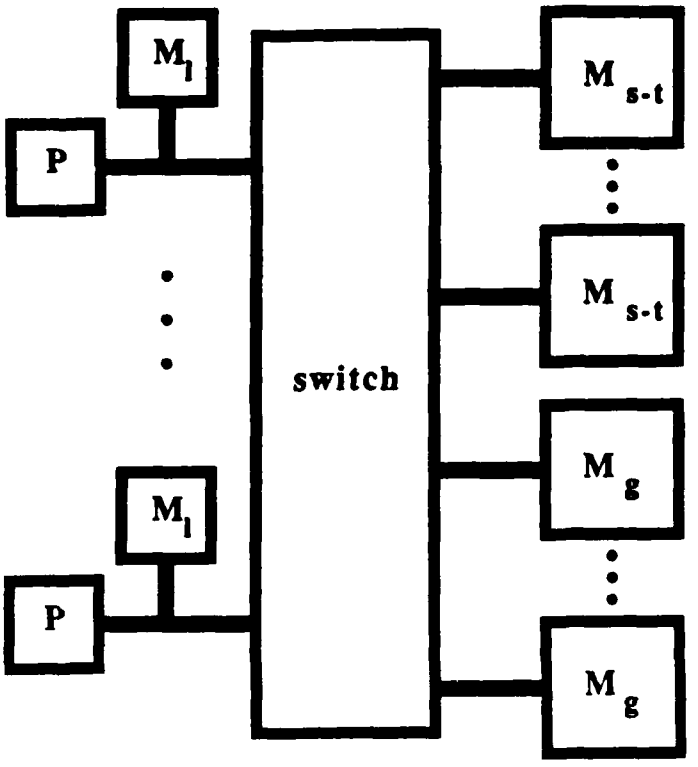

Figure 2: Block diagram of the Virtual Time Machine.

ables, and the VTM's scheduling queue(s) to the sequential simulator's event list. The execution mechanism is identical to the sequential one, except all processors concurrently execute the "remove next event and process it" loop that drives the simulation forward. Several events may be processed concurrently; the hardware automatically guarantees that the overall result is the same as if the events were executed sequentially. Because the hardware implements a simple generalization of the sequential discrete event simulation paradigm, one aspect of the Virtual Time Machine that is particularly attractive is its ability to *automatically* parallelize sequential simulation programs.

5 The Space-Time Memory System

A block diagram of the envisioned machine is shown in figure 2. Objects are stored in the space-time memory modules (M_{s-t}). Conventional RAM is used to hold code and local data (M_l), as well as certain global data structures such as the scheduling queues (M_g). The switch is conventional, e.g., an Omega network. The use of caches near the processors to buffer global data (not shown in figure 2) is currently under investigation.

The space-time memory system provides each task with a consistent, *global* view of the entire

program's state at any point in virtual time, just as is the case for sequential programs. This is especially useful in discrete event simulation where access to remote state variables is expensive in existing parallel simulation strategies.

Reading a variable from the space time memory at virtual time T returns the most recent (in virtual time) version of that variable written at virtual time T or less. A write at virtual time T either creates a new version at time T if none already exists, or overwrites the existing version. Here, accesses to the space-time memory can be made transparent to the application program because the temporal component of memory operations is always the virtual time of the currently executing task; the hardware can automatically supply the time component of the address by remembering the task's virtual time as it is removed from the scheduling queue.

5.1 Design Goals

The principal design goals regarding the implementation of the space-time memory are:

- *Fast access.* The access time for memory operations that do not trigger rollback should approach that of conventional memory systems.
- *Avoid excessive copying.* Copying the entire contents of the object to create a new version is *not* an acceptable solution.
- *Efficient memory utilization.* The aforementioned copying approach is very inefficient if tasks only modify a small portion of objects.
- *Rapid state restoration.* Rollback is an integral part of the computation mechanism, and occurs frequently enough that one *cannot* allow state restoration to be very expensive.
- *Exploitation of DRAM technology.* Conventional dynamic RAM should be used for the bulk of the storage used by the memory system. It would be very difficult for this approach to be economically competitive with conventional machines if it depended on a custom memory chip.

The proposed design meets these goals, except the second is only partially achieved. Although we perform very little copying when a new version is created, the storage reclamation process may require a significant amount of copying. However, storage reclamation is performed by dedicated

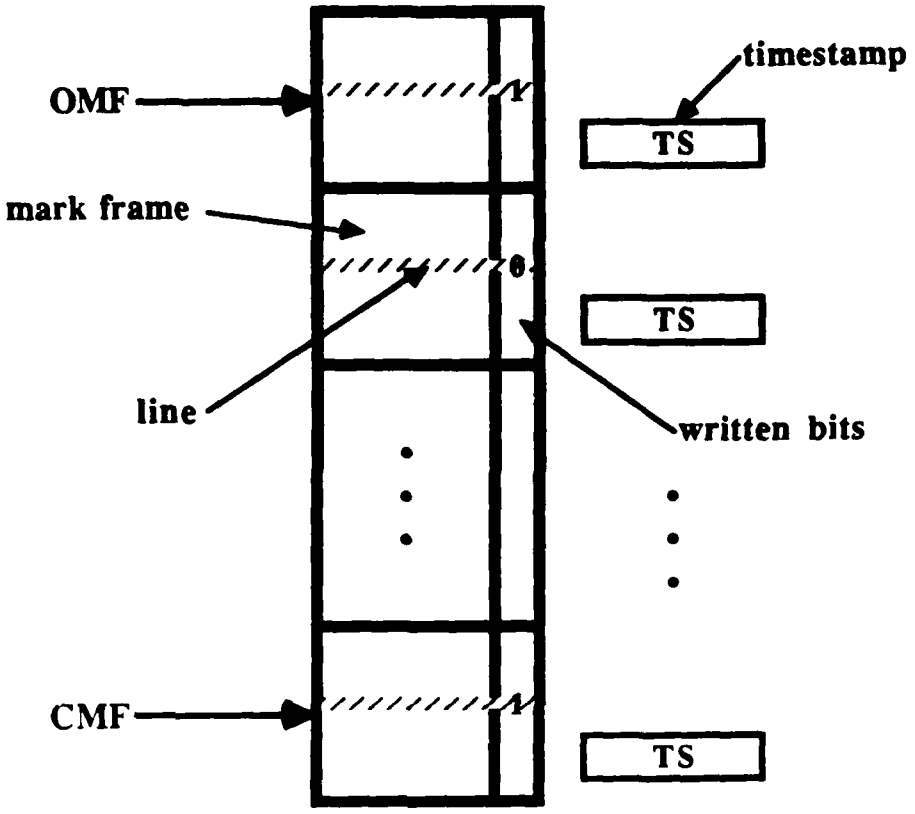

Figure 3: Memory organization for one object in the space-time memory.

hardware when the memory system would otherwise be idle, so we do not expect a significant performance degradation will result.

5.2 Implementation

We present an overview of the proposed implementation. This work is an extension of our earlier work on the *rollback chip*, which is described elsewhere [FTG88a].

The central data structure for implementing a single object is a stack of *mark frames*, depicted in figure 3. Each frame provides storage for a single version of the object. The *current mark frame* (CMF) register contains a pointer to the top of the stack, and the *oldest mark frame* (OMF) a pointer to the bottom of the stack. Each frame (version) also has a timestamp associated with it to indicate its temporal coordinate in the space-time diagram (figure 1), i.e., the virtual time of the write operation that caused the creation of that version. Creation of a new version is implemented by pushing a new frame on top of the stack (incrementing the CMF register). Rollbacks are implemented by popping and discarding some number of frames from the stack (resetting the CMF

register to the new top of stack frame).[9]

Each frame is further divided into some number of fixed length lines that are similar to the lines in a conventional cache memory system. Because no copying is performed when a new version is created, each frame normally contains "holes," i.e., lines in which no valid data has been written. Each line of each version has a single *written bit* associated with it indicating whether or not that line contains valid data.

Each frame has a fixed maximum size (e.g., 4 MBytes), and the stack contains a fixed number of frames (e.g., 256). The stack is actually managed as a circular queue, with creation of new frames accomplished by advancing the CMF pointer, and reclamation of storage by advancing the OMF pointer. Although it is possible to devise schemes to deal with stack overflow by allowing the stack to extend into a different area of memory, this adds a non-trivial amount of complexity to the control mechanism. A simpler (and probably just as effective) solution is to suspend execution of tasks that are far ahead in virtual time until the storage reclamation process has reclaimed adequate storage; such tasks are well ahead of others in the computation, so it is unlikely that they lie on the program's critical path.

Each memory read first converts the specified temporal address (T) into an integer index corresponding to the mark frame defined at time T, if such a frame exists, or the most recent frame older than T if none exists at T. A special component is described later that implements this translation function. The resulting frame number is used to log the memory access, as was described earlier. The spatial address selects a line and word within that frame. If the selected line's written bit is *not* set, the written bits for successive (older) mark frames are searched to find the location of the most recent version of the data. Once this frame is found, the number of that frame is *concatenated* with the spatial address to form the memory address that is used for the access. Finally, a special frame called the *archive frame* is defined that always contains the most recent version of the data that is older than the OMF. If no written bits are found in searching back through the mark frame stack, the required data are assumed to reside in the archive frame.

A write operation at virtual time T automatically creates a new frame if one does not already exist at that time, and sets the corresponding written bit. If the written bit was not previously set, the write must also locate the most recent version of data by searching the written bits as described above, because the write operation will not overwrite the entire line (assuming lines are larger than

[9]Actually, a few other operations are required to update the *rollback history* data structure, as described in [FTG88a], but these require very little time.

a single word). Writes may also initiate rollback, as described earlier.

5.3 The Space-Time Cache

Searching for the most recent version on each memory reference would be too time consuming to meet our performance goal. This problem is addressed by caching recently used lines of space-time data in a high speed memory. This cache lies within the space-time memory module, and only holds data for that memory module, so there are no coherence problems. Accesses for cache hits are very similar to that in conventional caches, so one can expect the space-time memory to achieve performance approaching that of conventional memory systems if a high hit rate is achieved.

A written bit search is only required on cache misses. However, even if misses are infrequent, they could significantly degrade performance if they are expensive. The miss time can be reduced to a manageable level by organizing the written bits so that a block of bits is read on each reference to the written bit memory; for example, our current design examines the written bits for 16 successive mark frames at each iteration of the search. Also, the written bits should be stored in fast static RAM, and the search can be pipelined to further improve performance. Still other techniques have been proposed to minimize the time required for searches on cache misses, but the techniques described above appear to be adequate [FTG88a].

The space-time cache does require some special operations. In particular, rollbacks may necessitate the invalidation of certain cache entries. Specifically, entries containing data for writes that were invalidated by the rollback must be purged. To implement this operation, frame number information is stored in the cache and embedded comparison logic is used to rapidly perform the invalidation.

5.4 Translating Virtual Time

The time coordinate used in the storage organization described above was a simple integer. Memory accesses generate arbitrary virtual times for the temporal address. Therefore, a mechanism is required to translate virtual time units to the integer time coordinate used by the memory system. This function can be performed in hardware using a *virtual time-translation component* that is similar to an associative memory. This operation could be performed on each memory reference. Alternatively, recently used translations could be stored in a small translation lookaside buffer.

Let us examine the translation mechanism for a single object. A register is associated with

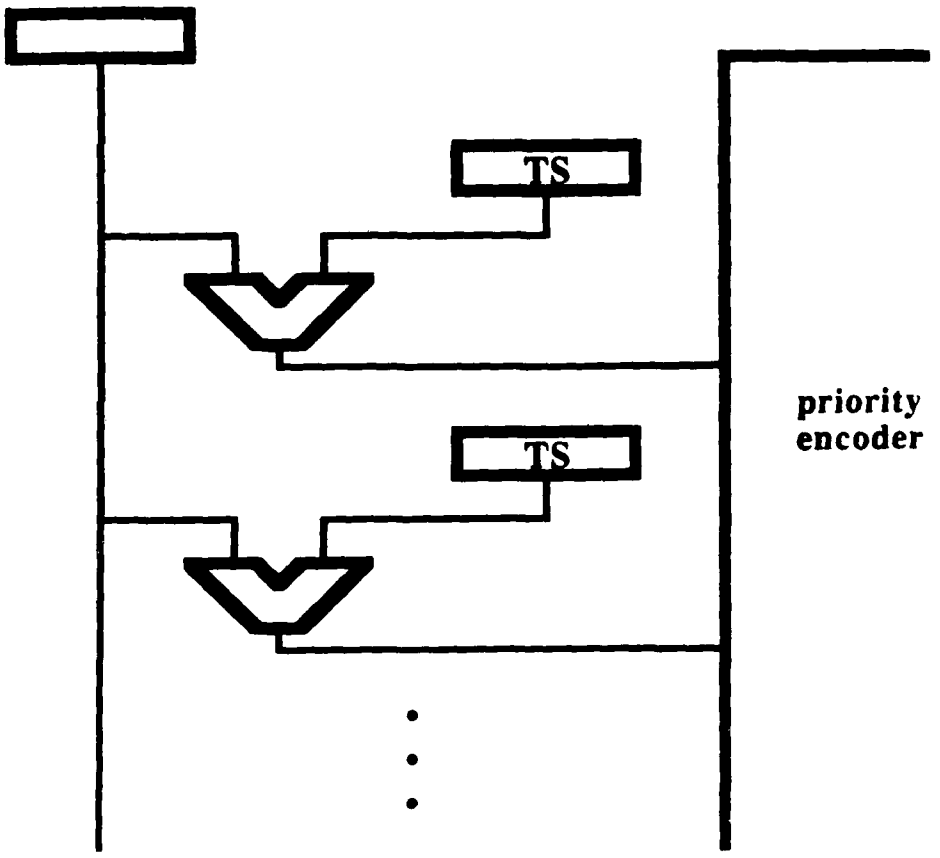

Figure 4: Virtual Time translation hardware.

each mark frame that indicates the virtual time of the write that caused the creation of that frame. Registers for successive stack frames contain strictly increasing values. Consider a read of the object at virtual time T. One must find the two registers R_i and R_{i+1} such that $R_i \leq T < R_{i+1}$, and return i as the temporal coordinate. This operation can be efficiently implemented by attaching a comparator to each register, and feeding the results into a priority encoder, as shown in figure 4. If virtual times are represented by floating point values, a representation should be selected that allows a simple magnitude comparator to be used (e.g., by placing the exponent in the most significant bits).

This approach can be easily extended to support multiple objects. To support n objects, each register is replaced with a RAM containing n memory locations. The translation process is identical to that described above, except that one must first use the object number to read each RAM (in parallel) before the comparison operation. Also, multiple copies of other object specific information (e.g., the OMF and CMF) must be maintained, one for each object.

5.5 Virtual Space-Time Memory

The implementation of the space-time memory system described above does not initially appear to meet our goal of memory efficiency. If each frame is 4 MBytes, and each stack contains 256 frames, each object requires a GByte of memory. In particular, if objects are large and only a small portion is modified by each task, the stack is very sparse, and most of this storage is wasted.

We attack this problem through the use of *virtual memory*. Each space-time address is obtained by concatenating the *translated* time coordinated (possibly after a written bit search) with the spatial coordinate, as described earlier. This address is then interpreted as a *virtual* address that is passed to a demand paging scheme. The paging mechanism only allocates physical memory to the version stack as it is needed, one page at a time. The fossil collection mechanism frees the pages when they are no longer required. Secondary storage is not strictly necessary unless we run out of physical memory pages. The space-time memory system is somewhat lavish in its use of address space, but stingy in its use of physical memory.

This approach need not require a microprocessor with an enormous address space; as mentioned earlier, the temporal address can be stored in an external register when execution of a task begins. Expansion of the spatial and translated temporal address occurs within the space-time memory. We expect that this approach can be implemented using existing 32 bit microprocessors.

Although virtual memory systems have been widely studied in the past, the strategy used here introduces a new wrinkle — both spatial *and* temporal locality translate directly to physical memory utilization. Consider a page of physical memory containing, say, 256 bytes. Figure 5 shows three methods of mapping a page of physical memory onto the space-time address space. In (a) the page is mapped onto a block of memory one byte wide, and 256 frames deep. Example (b) uses the opposite extreme — 256 bytes in space and only a single frame. The first approach exploits temporal locality, but completely ignores spatial locality. The second has the opposite flaw: it does not exploit temporal locality. No doubt the third approach (c) is better than the others because it simultaneously exploits *both* temporal and spatial locality in the application. Each of the three approaches is as easy to implement as the others — they only differ in the bits they select from the space-time address to determine the page number and offset, assuming the range of temporal and spatial addresses covered by a page is (for each) a power of 2.

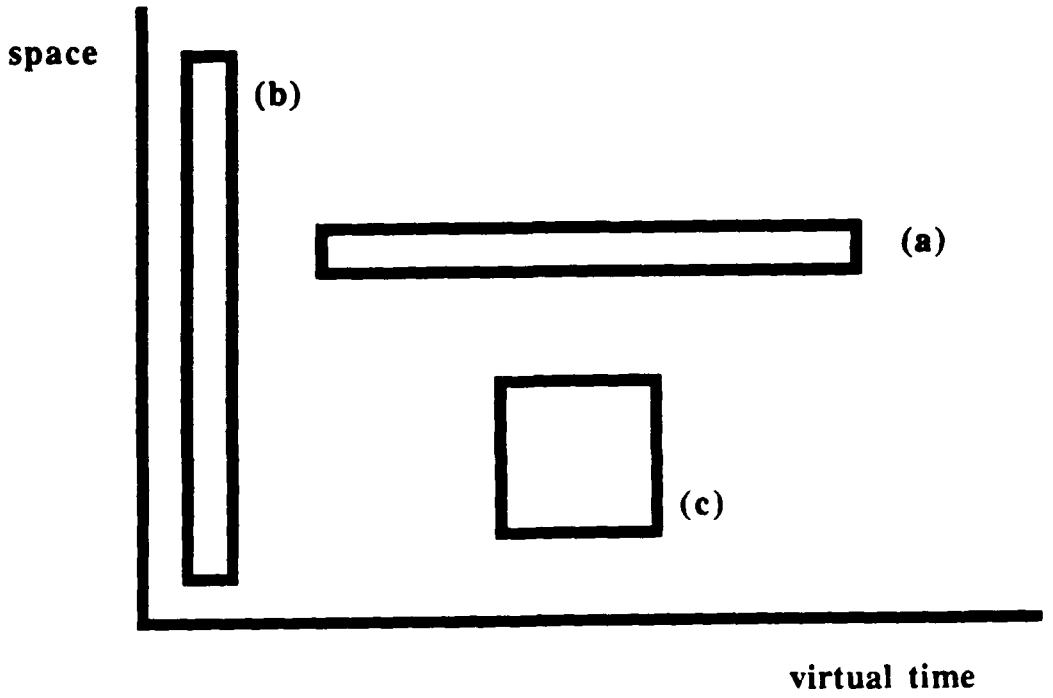

Figure 5: A page of physical memory in virtual space-time address space. (a) temporal locality only (b) spatial locality only (c) both spatial and temporal locality.

5.6 Storage Reclamation and Irrevocable Operations

The timestamp of the oldest task, for which processing has not yet been completed, provides a bound on the most distant rollback that can occur. This value is referred to as *global virtual time (GVT)* in Time Warp systems. GVT is necessary for *fossil collection*, i.e., reclamation of storage and commitment of irrevocable operations, e.g., I/O. Storage for versions older than GVT (except the most recent one) can be reclaimed. The most recent version of data in fossil collected frames must be copied to the archive frame as it may be needed for future memory references.

5.7 A Space-Time Memory Module

A block diagram of a single space-time memory module is shown in figure 6. A read request is first sent to the space-time cache, which includes the virtual time translation circuit described earlier. If a hit occurs, the data are simply read from the cache and returned to the processor. If a miss occurs, the written bit memory is interrogated to locate the most recent version, and a virtual address is constructed by concatenating the resulting frame number with the spatial

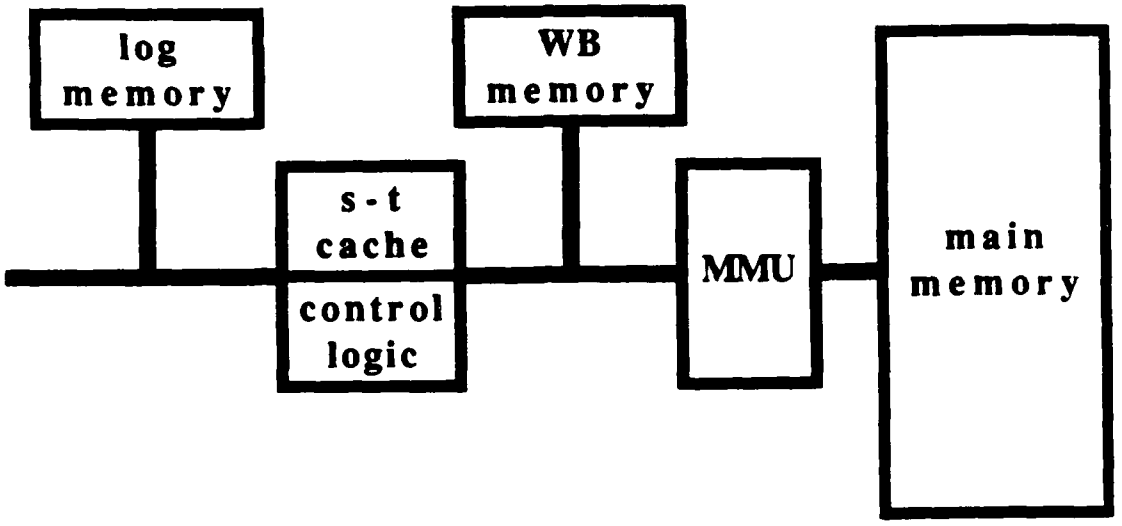

Figure 6: Block diagram of a single space-time memory module.

address. This address is translated by the memory management unit (MMU) to a physical address, the appropriate line is read from main memory, loaded into the cache, and the requested data are returned to the processor. Except for the written bit search, these steps are not unlike those in a conventional memory system. Finally, the log circuit is responsible for recording the memory reference. Logging can occur concurrently with the data fetch.

Writes may initiate rollback operations. If a write does trigger a roll back, a flag is returned to the processor indicating this fact. The processor, or alternatively, a special *rollback coprocessor*, performs the necessary rollback and cancellation operations.

6 Performance

Initial investigations have begun to evaluate the effectiveness of this approach. In particular, our near term goal is to demonstrate that the Virtual Time Machine provides a viable solution for attacking difficult, long standing problems in parallel discrete event simulation. Here, we summarize the central results that have been achieved thus far. Space does not permit a complete, in depth discussion of these studies, but, a more detailed treatment is presented elsewhere [Fuj87,Fuj89, FTG88a].

Perhaps the three most pressing questions regarding the viability of this approach are:

1. *Does the machine waste much of its time performing erroneous computations?* A rolled back

computation is equivalent to idle time in a conventional machine in the sense that no useful work is performed. If the computation spends most of its time executing computations that are eventually rolled back, performance will be poor.

2. *Is the overhead necessary to allow rollback, and the rollback mechanism itself, excessive?* History maintenance is required, independent of the frequency of rollback, so it must be relatively efficient. Also, we cannot assume that rollbacks are sufficiently infrequently that performance of the rollback mechanism itself doesn't matter.

3. *Can the space-time memory achieve adequate performance?* The space-time memory is the one unproven hardware component in the machine, so it must be demonstrated that it will provide adequate performance.

These questions relate to the efficiency of the two components of the computation model: the control mechanism that builds the execution history, and the state history mechanism. The first two questions are being investigated through a software based prototype implementation of the control mechanism on a conventional shared memory multiprocessor. The last question is being studied through simulations of the space-time memory. We describe each of these studies in turn.

6.1 Evaluations of the Control Mechanism

An implementation of a variation of the control mechanism on the BBN Butterfly^{TM} multiprocessor has been developed for experimentation with parallel discrete event simulation programs. Because this testbed does not have the benefit of special purpose hardware (in particular, the proposed space-time memory system), some restrictions are necessary. In particular, new versions of objects are created by brute-force copying of the entire object. Therefore, objects must be relatively small to prevent this overhead from biasing the results. Also, a process oriented model for simulation is used in which the parallel simulator consists of a collection of *logical processes* (LPs), each simulating a portion of the system being modeled. This is a very natural programming paradigm for simulating many types of systems, however, it does imply that tasks (or here, simulator *events*) only access a single object. Therefore, these experiments assume good spatial locality. Finally, we assume (conservatively) that a data dependence constraint violation will always occur whenever a process executes two events (tasks) out of their virtual time sequence. Detection of violations of data dependence constraints is cumbersome without the benefit of hardware support. The current

implementation also uses a static scheduling policy. Overall, the prototype is similar to Jefferson's Time Warp mechanism, with the addition of direct cancellation (which eliminates the need for anti-messages). Details of the implementation are described in [Fuj89].

In spite of these limitations, rollback and cancellation are fully implemented, so this implementation does provide a good means of measuring the overhead that arises from executing erroneous computations and performing cancellation and rollback. Moreover, it provides a lower bound on VTM performance for several interesting applications.

The benchmark programs that have been used thus far are simulations of closed queuing networks configured in a hypercube topology. A server is associated with each outgoing link. Simulations of communication networks would be expected behave similarly. A fixed number of jobs (messages) randomly circulate throughout the network. Incoming jobs are randomly routed to an outgoing link using a uniformly distributed random variable. The job service time is selected from an exponentially distributed random variable with mean of 1.0, and minimum value of 0.1.[10] The network topology contains many cycles and a moderate node degree, so ample opportunity for rollback is provided. On the other hand, it is also homogeneous and highly symmetric, and free from any inherent bottlenecks that might otherwise bias the results.[11]

Several experiments using different queuing disciplines were performed. In one simulator, jobs are assumed to be processed in first-come-first-serve order. In the second, some fraction of the message population is designated for modeling high priority jobs, while the rest have low priority. Jobs remain at the same priority level throughout the entire simulation. If a high priority job arrives while a low priority job is being serviced, the latter is *preempted*, and returned to its queue. Jobs within the same priority level are processed FCFS.

An important goal of this study was to illustrate that optimistic, rollback based approaches offer substantial benefits over conservative approaches in certain problem domains. The FCFS service discipline is a very favorable one for conservative approaches because it has good *lookahead* properties [Fuj88]. In contrast, the simulator with preemption has poor lookahead. By lookahead, we mean the degree to which the simulator can predict what will happen in the future based on what has already been simulated for the past.

Lookahead is crucial for achieving good performance in existing conservative synchronization

[10] *Results using a minimum value of 0.0 are virtually identical*; The 0.1 value was used to facilitate comparison with a version of a conservative simulation mechanism.

[11] Additional studies using heterogeneous, asymmetric workloads are also planned.

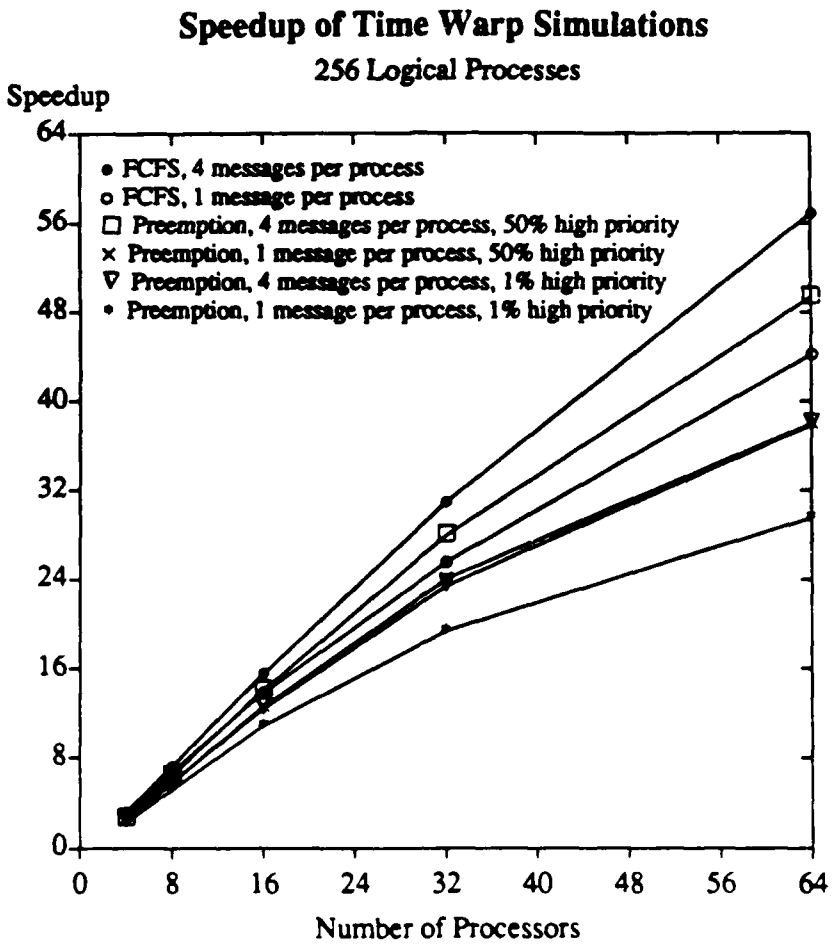

Figure 7: Speedup of 256 process hypercube simulator.

mechanisms. To see this, let us consider why the simulator with preemption is such a difficult case for a conservative synchronization mechanism. It is possible, albeit extremely unlikely, that a high priority message could affect every server in the simulation in a very small amount of simulated time. Conservative algorithms must guard against the worst-case scenario, so no matter how improbable, many processes will have to wait to make sure that the above scenario does not occur before they can proceed with the simulation. Further, because essential information is distributed across the entire system, determining when it is "safe" to proceed often introduces substantial overheads.

The experiments described here were performed on a BBN Butterfly™ multiprocessor that is housed at the University of Maryland. Numerous additional experiments using a smaller, albeit faster and locally available version of the Butterfly were also performed; these results are described in [Fuj89]. Figure 7 shows the speedup of the Time Warp simulator relative to an optimized

sequential simulator. The sequential simulator uses a splay tree to implement a global event list [ST85]; the splay tree has been reported to be a very efficient mechanism for implementing a priority queue [Jon86].

The speedup curves reflect performance when simulating an eight dimensional hypercube (256 processes) on Butterfly configurations containing as many as 64 processors. Message populations of 256 and 1024 (1 and 4 messages per process, respectively) are used in these experiments.[12] The experiments using preemption assume either 50% or 1% of the message population is high priority jobs. Performance using these benchmarks is encouraging. Speedups as high as 56.8 using 64 processors were obtained.

The speedup figures are especially encouraging when compared to performance of well-known conservative algorithms based on deadlock avoidance and deadlock detection and recovery [Mis86]. Benchmarks simulating 16 and 64 node hypercubes ran many times *slower* on eight Butterfly processors when using these message populations [Fuj89]. The Time Warp simulator achieved speedups ranging from 4 to 8 under these circumstances. These statistics demonstrate the advantages offered by optimistic synchronization methods over existing conservative mechanisms in certain problem domains.

Figure 8 shows the corresponding efficiency measurements for these experiments. Here, efficiency is defined as the number of events that were executed which were neither rolled back nor cancelled, divided by the total number of events that were processed. It provides an indication of the amount of time the simulator spent processing correct events relative to erroneous ones.[13] As one might expect, efficiency is highest when the problem is much larger than the hardware configuration. For the 64 processor case, efficiency figures for these benchmarks range from 57% to 81%. The benchmark using preemption yields somewhat lower performance because it contains less intrinsic parallelism. If a yet to be received message A is destined to preempt an already received message B, then the computation to determine B's departure time (and forward it to another processor) cannot be correctly performed until A has been received; this would not be the case if FCFS queues were used, or equivalently, if both messages had high priority.

These studies demonstrate that the overhead of rollback and cancellation and the degradation

[12]Memory constraints did not allow us to perform the *uniprocessor* simulation at higher message populations, but other experiments using a Butterfly with more memory per processor (but fewer processors) yielded similar performance results for higher message populations, e.g., 16 and 64 messages per process.

[13]Note, however, that efficiency is only a relevant measure if the problem contains a significant amount of parallelism; otherwise, poor efficiency is inevitable. Also, the efficiency figure defined here does not consider idle time.

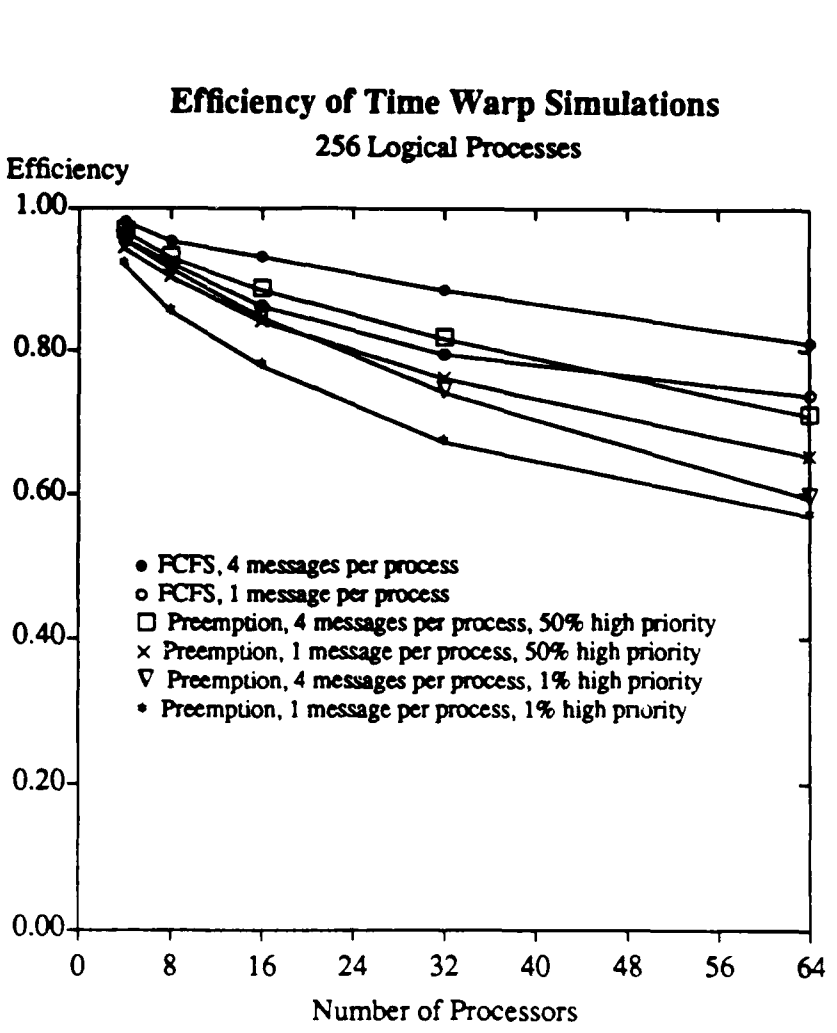

Figure 8: Efficiency of Time Warp simulations for 256 process hypercube (fraction of all processed events that were not rolled back or cancelled).

resulting from performing erroneous computations are not excessive for some problems of significant practical interest. Although a more comprehensive performance study of optimistic synchronization using Time Warp is required,[14] these initial performance measurements are encouraging.

6.2 Evaluation of the Space-Time Memory System

The space-time memory is a straightforward extension of the rollback chip (RBC). Results of extensive simulation studies of the RBC forms the basis of our claim that the space-time memory can achieve performance approaching that of conventional memory systems. Here, we summarize these results, after we first discuss the differences between the space-time memory and the RBC.

The rollback chip assumes that read references refer to the most recent version of data, so

[14] Such a study is currently in progress.

searches always begin from the frame on top of the version stack. Here, written bit search may begin in the middle of the stack. This necessitates the virtual time translation circuitry discussed earlier, and some minor changes in the design of the cache.

Also, the rollback chip required *explicit* generation of operations to create new versions and roll back the version stack.[15] Here, these operations are generated *implicitly* by the write operation. Generation of these operations can be implemented by the virtual time translation logic. Rollback is initiated if the timestamp of the write is less than the virtual time of the frame on top of the stack. Similarly, a new version is created if the write timestamp is greater than that of the top of stack frame. The translation logic already performs similar comparisons, so implementation of these functions is straightforward.

Because the mechanisms that are at the heart of the space-time memory are virtually identical to those used by the rollback chip, simulations of the RBC provide a good indicator of the performance of the space-time memory system. We describe below the simulations that have been performed for the RBC. Details of these studies are discussed in [FTG88a]. The central goal of this study was to determine the average access time for read and write operations to the space-time memory (excluding the time to do the actual rollback, which was examined in the Butterfly experiments). Creation of a new version and state restoration after rollback require a constant amount of time, e.g., one or two clock cycles.

The rollback chip's performance was compared to a traditional memory system containing a cache with identical parameters as the rollback chip's cache (in size, organization, etc.). There are two sources of performance degradation in the rollback chip relative to the conventional system:

1. The RBC cache will yield a lower hit rate than the conventional cache because rollbacks may necessitate invalidation of certain cache entries.

2. Miss penalties in the space-time cache will be larger because a search to find the most recent version is required.

Workloads to drive the simulation were generated stochastically. Addresses for memory accesses are created from a normally distributed random variable. The mean of the address trace distribution is periodically changed to simulate phase changes in the program. Operations to create new versions, roll back the computation, and reclaim storage were also inserted into the stream of read and write

[15] It was assumed that the processor generated these operations by writing into the RBC's control registers

operations. Numerous rollback scenarios were examined, ranging from frequent, short rollbacks to infrequent, long ones.[16] Rollback distances were selected from a negative exponentially distributed random variable.

These factors are combined into a single parameter which we call the *event rate*. The event rate indicates the rate at which the computation is advancing in virtual time. More precisely, if F_{NV} is the frequency at which new versions are created on top of the stack, F_{RB} is the frequency at which rollbacks occur, and RB_{dist} is the average rollback distance (number of versions popped from the stack), then the event rate is defined as $F_{NV}/(F_{RB} * RB_{dist})$. It indicates the net rate of progress for the computation (e.g., two steps forward for every step back). In our experiments on the Butterfly, situations where event rates were as low as 2, or as high as 20 or more have been observed for programs that still achieved a respectable speedup.

The rollback chip simulations indicated that the cache incurs a hit rate degradation that can be expected to be below 2%, and typically is less than 0.50%. The search through written bits to locate the most recent version of data typically required two or three references to the written bit memory, where each reference returns the written bits for 16 versions. In order to reduce the miss penalty further, the written bits should be stored in a dedicated static RAM. Under these circumstances, the miss penalty is estimated to be approximately two to three times that for a conventional cache.

Using the simulator to derive hit rate degradation and miss penalty values, one can derive the estimated access time for the rollback chip as $P_{hit}T_{cache} + (1.0 - P_{hit})T_{memory}$ where P_{hit} is the probability of a cache hit (including the aforementioned degradation in hit rate), and T_{cache} and T_{memory} are the access times to the cache and main memory respectively (T_{memory} includes the additional miss penalty). In addition, not all memory references will access the space-time memory: instruction fetches and references to local data access conventional RAM, making the performance of the space-time memory less critical than it would otherwise be.

The curves shown in figure 9 show the overall performance degradation for the rollback chip's memory relative to the conventional system as a function of the hit rate in the conventional cache. The upper set of curves assumes that 25% of all memory references utilize the rollback chip, and the lower curves assume 10%. In the former case, there are an average of 20 read and write operations before a new version is created or rollback occurs, and in the latter case, an average of

[16] It is not possible to have frequent, long rollbacks because the computation would then be going backwards, which is provably impossible.

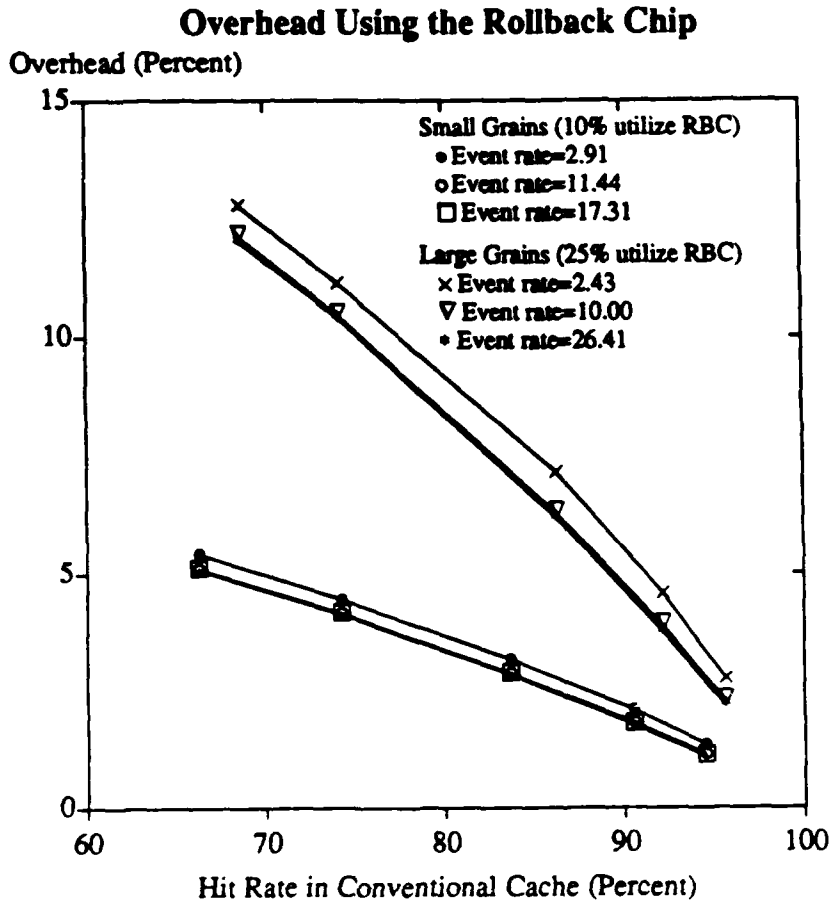

Figure 9: Overhead using the rollback chip.

200 operations. These correspond to "small grained" and "large grained" tasks, respectively. The average rollback distance is between two and three versions, i.e., rollbacks are short, but can occur frequently. Performance of the cache is generally better if rollbacks are long, but infrequent.

As can be seen, performance approaches that of the conventional memory system for high hit rates. This is because rollback chip penalties only arise on misses. Modern caches routinely achieve hit rates exceeding 90%, so we anticipate the overall performance penalty will only be a few percent in most situations.

7 Automatic Parallelization of Sequential Programs

An important application in which the Virtual Time Machine may offer significant advantage over existing parallel computers is in the automatic parallelization of sequential programs. Here,

we outline the basic approach, and point out some important questions that have not yet been resolved.

First, the computation must be subdivided by the compiler to define the tasks that will be created as the computation unfolds. For example, the tasks might be individual iterations of a for-loop, the execution of a procedure, a block of sequential code, etc. Also, the variables that persist from one task to another must be mapped to objects, as was described earlier.

The tasks must also be mapped onto an appropriate virtual time scale to allow detection of data dependence violations. The timestamps should more or less reflect the relative order in which the tasks would have been executed by a sequential machine. The method for assigning virtual times to tasks is currently an open question. To see where the difficulty lies, consider a while-loop W that is immediately followed in the sequential program by another computation, e.g., a procedure call P. Let us assume that a task is created for each iteration of W, and a single task is created for P; it may be desirable for all of these tasks to execute concurrently. Each iteration of W must be assigned a timestamp that is less than P's timestamp. However, the number of iterations that will be executed by W is not known until the loop is fully executed. Therefore, it is not clear how large to make P's timestamp.

Jefferson has proposed reusing timestamps to attack this problem [Jef88]. In the above example, this would amount to assigning multiple iterations of the while-loop to a single task. For example, if we decide we only want to create at most 100 concurrent tasks for executing the while-loop, then we could assign iterations $1, 101, 201, \ldots$ to the first task, $2, 102, 202, \ldots$ to the second, and so on. An additional barrier synchronization is required after each set of 100 iterations is completed to ensure correctness. Unneeded tasks in the last set of 100 iterations would be canceled once the terminating condition of the loop was determined. Also, we note that Tinker has proposed another approach for attacking this problem that uses a two-level timestamp [Tin88].

Another important question that must be resolved is dealing with conditionals. A straightforward solution is to partition the program so that conditionals are encapsulated within a single task. Alternatively, one might predict the outcome of the branch, and create a new task based on this prediction (this is what was done in the while-loop example described above). Branch prediction has been successfully used in pipelined computers [LS84].

One special case of the automatic parallelization problem for which the Virtual Time Machine approach may be especially well suited is the automatic parallelization of sequential discrete event

simulation programs. Here, the problems of task definition and assignment of virtual times have trivial solutions. The most important, unresolved issue that remains is the method for partitioning the state space of the simulation into data objects. Some modification of the sequential simulator may be required if the application program is poorly structured. For example, if all events modify a single global state variable, accesses to this variable will force serialization of the entire program. However, because simulations often model systems where there is a substantial amount of locality of effect (i.e., a single event seldom has an immediate effect on the state of the *entire* system), there is some reason to suspect that some success can be achieved in this problem domain.

Finally, one other important question merits some mention. The scheduling and dynamic load balancing policy plays an important role in the performance of the machine, more so than in conventional parallel computers. This is because the scheduling mechanism has a great impact in determining the frequency of rollback. The machine should always attempt to execute those tasks that are least likely to be rolled back. In the absence of any additional information, preference should be given to tasks with small timestamps; the task that has the smallest timestamp in the entire system is guaranteed not to be later rolled back. Also, if caches are used to buffer space-time memory near the processor, the task to processor assignment should also attempt to maximize locality of reference.

8 Conclusion

The Virtual Time Machine architecture offers a new approach to parallel computation by providing hardware to detect violations of data dependence constraints at runtime, and using a rollback mechanism to automatically recover from them. Initial performance studies indicate that this approach offers good potential for speeding up large-scale discrete event simulation problems that cannot be solved using existing machines. Further, we believe that this approach shows promise for implementing automatic parallelization of sequential discrete event simulation programs.

The techniques used by the Virtual Time Machine have wide application in the more general realm of parallel computation. Research in optimistic, rollback based computation is still in its infancy, so it is difficult to judge the overall effectiveness of this approach for arbitrary applications. However, we feel that, when combined with sophisticated compiler technology and appropriate scheduling and load balancing mechanisms to minimize the frequency of rollback, this approach can lead to new inroads into difficult problems such as the automatic parallelization of sequential

programs.

A key characteristic of the Virtual Time Machine is its explicit representation of temporal aspects of the computation as an integral component of the machine architecture. This reflects our belief that many difficult problems in parallel computation today stem from this deficiency in existing machines. We contend that this is the reason parallel discrete event simulation has remained such a difficult problem over the last decade, and is a critical limiting factor in the success that can be achieved in automatically parallelizing sequential programs. Finally, we note that it is interesting that much of the current work in parallel program *debugging* involves constructing histories similar to those which are automatically managed by the Virtual Time Machine hardware [LM87,CMN88,GF88]. We feel that many important advantages of this approach have yet to be explored.

9 Acknowledements

Jya-Jang Tsai added instrumentation code to the rollback chip simulator and collected much of the data reported in figure 9. The author also wishes to thank David Jefferson and Peter Tinker for their comments and suggestions.

References

[CMN88] J. D. Choi, B. P. Miller, and R. Netzer. *Techniques for Debugging Parallel Programs with Flowback Analysis.* Technical Report #786, Computer Science Department, University of Wisconsin. August 1988.

[CUL88] J. Cleary, B. Unger, and X. Li. A Distributed And-Parallel Backtracking Algorithm Using Virtual Time. *Proceedings of the SCS Multiconference on Distributed Simulation*, 19(3):177–182. July 1988.

[FTG88a] R. M. Fujimoto, J. Tsai, and G. Gopalakrishnan. *Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp.* Technical Report UUCS-88-011, Dept of Computer Science, Univ. of Utah, July 1988.

[FTG88b] R. M. Fujimoto, J. Tsai, and G. Gopalakrishnan. Design and Performance of Special Purpose Hardware for Time Warp. *Proceedings of the 15th Annual Symposium on Computer Architecture*. June 1988.

[Fuj87] R. M. Fujimoto. *Performance Measurements of Distributed Simulation Strategies.* Technical Report UU-CS-TR-87-026a, Dept. of Computer Science, University of Utah, Salt Lake City. November 1987.

[Fuj88] R. M. Fujimoto. Lookahead in Parallel Discrete Event Simulation. *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988.

[Fuj89] R. M. Fujimoto. *Time Warp on a Shared Memory Multiprocessor.* Technical Report UU-CS-TR-88-021a, Dept. of Computer Science, University of Utah, Salt Lake City, 1989.

[Gaf88] A. Gafni. Rollback Mechanisms for Optimistic Distributed Simulation Systems. *Proceedings of the SCS Multiconference on Distributed Simulation*, 19(3):61–67, July 1988.

[GF88] A. J. Gordon and R. A. Finkel. Handling Timing Errors in Distributed Programs. *IEEE Transaction on Software Engineering*, 14(10), October 1988.

[Jef85] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[Jef88] D. R. Jefferson. private communication. November 1988.

[Jon86] D. W. Jones. An Empirical Comparison of Priority-Queue and Event-Set Implementations. *Communications of the ACM*, 29(4):300–311, April 1986.

[Kni86] T. Knight. An Architecture for Mostly Functional Programs. In *Proc. Lisp and Functional Programming Conference*, ACM, Cambridge, Mass., August 1986.

[KR81] H. T. Kung and J. T. Robinson. On Optimistic Methods of Concurrency Control. *ACM Transactions on Database Systems*, 6(2), June 1981.

[Kuc78] D. Kuck. *The Structure of Computers and Computation.* Volume 1, John Wiley & Sons, New York, 1978.

[Li88] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. *Proceedings of the 1988 International Conference on Parallel Processing, Vol. 2*, 94–101, August 1988.

[LM87] T. J. Leblanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4), April 1987.

[LS84] J. Lee and A. J. Smith. Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Computer*, 17(1), January 1984.

[Mis86] J. Misra. Distributed Discrete Event Simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[Pfi82] G. F. Pfister. The Yorktown Simulation Engine: Introduction. In *Proc. 19th Design Automation Conference*, pages 51–54, June 1982.

[ST85] D. D. Sleator and R. E. Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[Tin88] P. Tinker. *A Model for General Rollback Computing.* Technical Report, Rockwell International, December 1988.

[TK88] P. Tinker and M. Katz. Parallel Execution of Sequential Scheme with ParaTran. In *Proc. Lisp and Functional Programming Conference*, ACM, Snowbird, Utah, July 1988.